Master Thesis:

Access Control in the Internet of Things

Author – Denis Sitenkov
Supervisors – Ludwig Seitz,
Shahid Raza,
Göran Selander

Abstract

The new generation of Wireless Sensor Networks, that is known as the Internet of Things enables the direct connection of physical objects to the Internet using microcontrollers. In most cases these microcontrollers have very limited computational resources. The global connectivity provides great opportunities for data collection and analysis as well as for interaction of objects that cannot be connected to the same local area network. Many of application scenarios have high requirements to security and privacy of transmitted data. At the same time security solutions that are utilized for general purpose computers are not always applicable for constrained devices. That leaves a room for new solutions that takes into account the technological aspects of the Internet of Things.

In this thesis we investigate the access control solution for the IETF standard draft Constrained Application Protocol, using the Datagram Transport Layer Security protocol for transport security. We use the centralized approach to save access control information in the framework. Since the public key cryptography operations might be computationally too expensive for constrained devices we build our solution based on symmetric cryptography. Evaluation results show that the access control framework increases computational effort of the handshake by 6.0%, increases the code footprint of the Datagram Transport Layer Security implementation by 7.9% and has no effect on the overall handshake time. Our novel protocol is not vulnerable to Denial of Service or Drain Battery Attack.

Table of Contents

1 Introduction

The term The Internet of Things (IoT) is commonly used to name a set of objects (or things) that are directly connected to the Internet using the Internet Protocol (IP) stack. That is the main difference of wireless sensor networks (WSN) of previous generation where nodes were organized in a local network with special protocols like ZigBee [34] or WirelessHART [35]. Connection of objects to the global network in the IoT opens the opportunity for global data analysis. Typical applications for the IoT are home automation (e.g. smart home), personal health monitoring (e.g. measurements of heart rate, pulse or temperature), building automation (e.g. control heating, electrical and ventilation systems of the building), industrial automation (e.g. control of the electrical grids) and smart cities.

An example of the resource-constrained microcontroller is shown in Figure 1. This is CM5000 that is based on an open source TelosB mote platform [7]. The hardware is based on the low power 16 bit microcontroller from MSP430 family. This microcontroller has only 48KB of program memory and 10 KB of data Random Access Memory (RAM). It uses IEEE 802.15.4 2.4GHz wireless module. This node is powered with 2 AA batteries (3V in total). In general case these nodes are supposed to work for few years without battery replacement.

Figure 1.1. TelosB mote CM5000 [7].

Use cases mentioned above define specific network activity of devices in the IoT. These devices usually exchange with small packages so high network bandwidth is not as important as for multimedia applications. Moreover multicast transmission is more important for machine-to-machine communication than reliable transport.

Objects in the IoT (the same as in WSN) are controlled via microcontrollers that are constrained in computational power, memory space and, often, in power consumption. At the same time protocols that are used by general purpose computers like Transmission Control Protocol (TCP) and HyperText Transfer Protocol (HTTP) are too resource consuming to be used on highly constrained devices. Moreover, IEEE 802.15.4 radio protocol that is widely used in WSN has a limited Maximum Transmission Unit (MTU) that does not

meet the MTU required by IP protocol version 6 (IPv6). These limitations lead developers to use a special protocol stack for the IoT.

The specific protocol stack, limited bandwidth of WSN, constrained resources of devices that control objects, and specific use cases in the IoT define specific requirements for security solutions. From one side these solutions should be as reliable and secure as ones for other internet applications but at the same time have less computational complexity, have smaller packages to avoid fragmentation, and less communication overhead. Furthermore, these security protocols and algorithms should be able to run over the unreliable transport and support multicast communications.

The specific protocol stack, general usage scenarios as well as constrained resources of the nodes require different security solutions. For example on the transport level the Datagram Transport Layer Security (DTLS) protocol (extension of Transport Layer Security (TLS) protocol over unreliable transport) can be used [4]. DTLS as well as TLS is a standard solution for authentication, key exchange, and secure communication. However, the DTLS protocol does not take into account the limited computational resources of WSN nodes. Moreover, it doesn't suggest any solution for authorization problems. The possible solution for authorization and access control is the main focus of this thesis.

1.1 Contributions

This work suggests the lightweight, fast and secures access control protocol that can be used on constrained devices in the IoT. Contributions of the thesis are :

- The design and evaluation of the access control framework for constrained devices that is built on top of the industrial security protocols,
- Recommendations for possible use cases of the designed access control protocol.

1.2 Outline

This thesis is structured in following order. Chapter 2 gives an overview of existing network protocols stack in the IoT and security protocol that can be used on the transport layer. Chapter 3 describes in detail the problem that we are trying solve and the goal that we are trying to achieve. Chapter 4 covers the state-of-art solutions for access control protocols in the IoT. In Chapter 5 we formulate our solution for the problem and cover implementation details in Chapter 6. Chapter 7 contains evaluation results, analysis and improvements for our solution. In Chapter 8 we conclude the thesis and give consideration of future work.

2 Theoretical backgrounds

Protocol stack that is used in IoT is different from one that is used in the most of internet applications. The difference of protocols for each layer are shown in the Table 2.1

Table 2.1

Layer	**IoT**	**General applications**
Application layer	CoAP, CoAPs	HTTP, HTTPs
Transport Security layer	DTLS	TLS
Transport layer	UDP	TCP
Network layer	IPv6	IPv6, IPv4
	6LoWPAN	
Link layer	IEEE 802.15.4	IEEE 802.3, 802.11

2.1 IPv6 over Low power Wireless Personal Area Network (6LoWPAN)

The IEEE 802.15.4 radio protocol that is widely used in WSN on the link layer, has 127 bytes MTU [1]. At the same time IPv6 standard requires MTU to be at least 1280 octets [2]. In spite of the fact that applications that utilize IEEE 802.15.4 link layer in most cases have a very limited payload size, there still can be cases when the IP packet cannot be transmitted without fragmentation. To overcome this issue 6LoWPAN defines the new layer above the IEEE 802.15.4 link layer and below IP layer. In order to achieve maximum performance the 6LoWPAN standard defines two important concepts [2].

- Header compression that defines the way to compress IP and UDP headers to minimize the payload.
- Fragmentation that defines an algorithm to fragment and assemble IP packets that are larger than IEEE 802.15.4 MTU.

2.2 User Datagram Protocol (UDP)

In internet applications that utilize the request-response pattern it is common to use the TCP protocol on the transport level. This protocol has significant benefits as it guarantees delivery of packages as well as order of delivery. At the same time it requires a communication overhead for connection, additional resources to maintain the connection state and package delivery confirmation, which may cause timeouts due to the IEEE 802.15.4 network latencies. Properties mentioned above are significant shortcomings for the

application of TCP in the IoT. Moreover, many applications in the IoT do not benefit from guaranteed package delivery. As a result, UDP has become the preferred transport layer solution for the IoT. If order and guaranteed delivery is required it becomes a part of application level protocol such as CoAP.

2.3 Datagram Transport Layer Security (DTLS)

The main idea of the DTLS protocol is to use Transport Layer Security (TLS) over an unreliable datagram transport layer. DTLS features the same 4 sub protocols as TLS [4].

- The Handshake protocol defines authentication, key exchange, and setting up the reliable connection.
- The Record protocol defines secured communication.
- The Alert protocol defines error notification.
- The ChangeCipherSpec protocol contains only one message which is logically a part of handshake process but separated into a different protocol.

At the same time DTLS is different because of unreliable datagram transport. The potential message loss is also a reason to maintain its own retransmission timer. Possible message reordering forces DTLS to maintain a sequence number for messages and explicitly include this number in the header. In order to reduce the Round Trip Time (RTT) messages can be grouped into flights.

The successful DTLS handshake for symmetric keys is shown in the table 2.2.

Table 2.2. DTLS handshake.

Flight	**Client**	**Server**
1	ClientHello	
2		HelloVerifyRequest
3	ClientHello	
4		ServerHello ServerKeyExchange* ServerHelloDone
5	ClientKeyExchange ChangeCipherSpec Finished	
6		ChangeCipherSpec Finished

The client starts the handshake by sending a ClientHello message (Flight 1). As DTLS operates over the connectionless UDP protocol it has a protection against Denial of Service (DoS) attack. DTLS generates the cookie that is an HMAC of a randomly generated secret, client IP address, and ClientHello parameters and sends it to the client in HelloVerifyRequest (Flight 2). Client in return repeats the same ClientHello message with cookie included (Flight 4).

When server receives ClientHello with cookie it verifies the cookie and if it is valid generates random number and sends it in ServerHello message with ServerHelloDone (Flight 4). Client, in its turn, generates master secret and sends ClientKeyExchange that contains psk_identity, ChangeCipherSpec, and Finish message that contains all previous messages encrypted with generated master secret (Flight 5). The psk_identity field identifies the key that server should use for a handshake. It is used only in PSK mode. Upon receiving Flight 5 server verifies the content of the Finish message and replies with ChangeCipherSpec, and Finish messages (Flight 6).

2.4 Constrained application protocol (CoAP)

On the application layer HTTP is commonly used for most applications that implement a client/server model. At the same time this protocol is supposed to run over reliable transport (like TCP) and cannot be used over UDP. Moreover running HTTP protocol may require too much computational resources (like parsing HTTP headers, form parameters end, etc) for a constrained device and does not take into account models that are used in the IoT (e.g. multicast, and unconfirmed requests). In order to overcome these issues a new protocol CoAP [3] is being designed generally as a subset of HTTP protocol that can be used over UDP transport. CoAP takes into account constrained computational resources of microcontrollers and scenarios of machine-to-machine communications [3]. The main conceptual difference of CoAP from HTTP is a message abstraction that determines the type of request or response.

- A confirmable message request is sent when client is supposed to get a response or delivery confirmation. The response can be an acknowledgement message or non-confirmable message or both of them.
- A non-confirmable message request is send when client does not expect a confirmation of request.
- An acknowledgement message is sent as a response to confirm that request was delivered. It may contain additional data or be empty.
- A reset message is sent as a response to confirmable or not confirmable node to notify that request was received but some data was missed. Usually it happens when server node was restarted.

CoAP draft recommends using DTLS protocol to secure network traffic. In this case DTLS is running above transport layer (UDP) and below the CoAP. In

addition to NoSec mode when no security support is provided, following security modes are defined.

- Pre-Shared Keys [5]. In this mode symmetric keys are provided to client and server before the start of DTLS handshake. It is based on symmetric cryptography.
- Raw public key certificates [6]. This mode utilizes asymmetric cryptography algorithms (in particular Elliptic Curve Cryptography). Client and server are supposed to be provided with asymmetric key pair but not in X.509 format
- X.509 Certificates mode is the same as raw public key certificates with only difference that public keys are supposed to be provided in X.509 format.

2.5 Alternative approach over UDP

A different protocol stack that can potentially replace CoAP/DTLS in some IoT applications that are supposed to work with streams of data is being developed by Google. This protocol stack includes SPDY [9] protocol that is supposed to replace HTTP protocol. SPDY is supposed to be used over TCP but Quick UDP Internet Connection (QUIC) [10] protocol is being developed to replace TCP and improve streams multiplexing.

To secure QUIC connections Google offers a new QUIC Crypto [11] protocol. QUIC Crypto is based on TLS protocol but at the same time is designed to optimize handshake round trip time (RTT) and has following main differences.

- No PSK mode is specified. Only asymmetric cryptography is used and keys are provided as X.509 certificates.
- The key exchange is based on the Diffie-Hellman protocol but server does not generate a new random number for each handshake. The server number is stored in server config and remains the same for a long period of time.
- Ciphersuite is also predefined in server config (for now only AES-GCM is available).

At the same time the goal of QUIC and QUIC crypto is to optimize the bandwidth rather than to minimize resources utilization for constrained devices. For example 0-RTT handshake is supposed to RSA digital signatures which can be resource consuming and long messages may cause package fragmentation in 802.15.4 networks.

3 Problem statement

As mentioned above devices that are connected to the IoT are highly constrained in memory and computational power. Moreover, 802.15.4-based networks are lossy, have low bandwidth and high latency. In this case, maintaining access control policy information inside these devices can be difficult and evaluating access control requests may be expensive (especially if it requires network lookup). The situation becomes even worse if the number of potential clients that have an access to a device is high and these clients are dynamically changing, hence control policies cannot be statically preconfigured in nodes. Hence, the access control information should be managed in some server outside of the device with minimum direct communication between the server and device. At the same time, access control for the internet connected devices that utilize CoAP protocol can have 2 levels of granularity.

- Device level access, when any agents that have remote access to the device have full access rights to it resources.
- Resource level access, when each request is supposed to be verified against access control information.

As each node in IoT often perform very specific function (e.g. remote temperature sensing, control of a power switch, etc.) resource level access control is not always necessary and device level control is often enough.

As mentioned above, CoAP specification defines the utilization of DTLS protocol for authentication and secure communication. So DTLS can also be used for solving the device level access control problem. If DTLS connection can be established then agent is granted full access to the device resources otherwise access is denied.

CoAP defines 3 possible secure modes [3] - Pre-Shared Keys (PSK) [5], Raw public key (RPK) [6], X.509 Certificates. In X.509 Certificate mode a sensor node is provided with one or more public keys that can be used to verify previously unknown X.509 certificate. From one side this mode makes possible to manage the dynamic access to the node without key reprovisioning by issuing signed certificates to authorized agents. But from the other side the certificate cannot be withdrawn before it is expired. The certificate expiration check requires the device to be able to synchronize and maintain the real time clock that is not always possible for a constrained device, Moreover this mode can be too resource consuming for highly constrained devices because of the size of certificates and time for signature verification.

At the same time in PSK or RPK modes keys are supposed to be provisioned to device. Keys can be provisioned in a secure way, either manually or remotely via secure channel. At the same time in some applications (like pay-per-use systems) the fast dynamic access management is a crucial requirement. Therefore new ways for dynamic authorization managements should be designed, evaluated, and analyzed.

The main goal of this thesis is to define the access control framework that can be built on top of the DTLS protocol running in PSK mode and evaluate this framework of the hardware platform that have limited computational resources.

4 Related work

Previously, various access control solutions that allow the dynamic access management have been proposed for different IoT application scenarios.

Zhang et. al. [12] propose the solution for dynamic access allocation. In this approach the device owner provides clients with one time token that can be used to access device in the network. This paper describes different approaches to perform token reuse detection. Some of these approaches involve replication of reused tokens and some suggest distributed token storage. Also, a defense for possible man in the middle attack is suggested. At the same time this access token verification is based on verification of digital signature using RSA algorithm that may be time and resource consuming for constrained devices.

The proposal described in [13] applies usage control model for the IoT. This approach maps the UCON abstractions to IoT entities and is based on fuzzy theory. Unfortunately there are only few experiment present that does not provide enough data of evaluation of the approach on IoT nodes.

The work presented in [14] demonstrates delegated capability based approach and based on UDP and CoAP protocols. The general principle is shown on Figure 4.1.

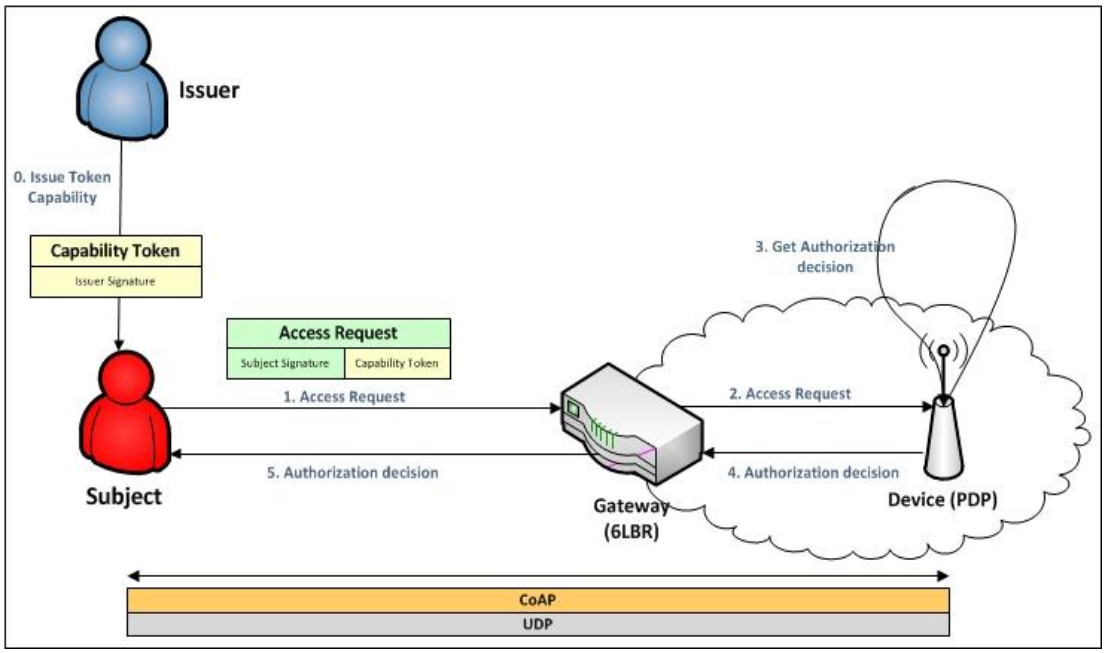

Figure 4.1. Distributed capability-based approach [14]

Access tokens are provided by issuer to a client with ECC digital signature in JavaScript Object Notation (JSON) format. A token contains information about resource to access, action that can be executed and additional conditions that is supposed to be checked by device. Server verifies digital signature and performs operation if permissions are granted by token.

Authors also evaluated their approach and measured that each operation requires 480.96ms in total and 89.1% is dedicated for cryptographic computations. Taking into account that access control check is required for each request, this overhead is quite big in terms of power efficiency and computational time.

4.1 Lightweight Machine to Machine

None of the papers mentioned above addresses the authentication and communication security problems. An interesting access control solution is being developed by Open Mobile Alliance (OMA). OMA is a standards body that develops protocols and standards for mobile devices and cellular networks. The lightweight machine to machine protocol (LWM2M) is described in [15]. It includes authorization, authentication, and channel security protocols definition. Moreover it was designed to support CoAP as an application level protocol. In this approach UDP is supposed to be secured with DTLS but 2 additional ciphersuites must be supported by implementation:

- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256

The security policy of the device is managed by centralized Bootstrap Server. The access control solution is described in details. It is defined in terms of Server Account (SA), Resource, Object and Object entity, Access Control Object and Access Control Object Instance. Clients are authenticated based on their SA. SA may be provisioned manually or via Bootstrap Server. Resources are entities that are supposed to perform different operations on Objects or Object Instances and liked with Objects or Object Instances. The initial provisioning grants to certain clients the right to perform the create operation on the certain Object and create a new Object Instance. The client that creates object instance is an Object Owner and can grant or revoke privileges to perform different operations on Object to other clients.

In spite of the fact, that protocol [15] is very well defined and supports different levels of access control and flexible privileges management it may be too complicated to be implemented in constrained devices. Moreover the centralized key provisioning from bootstrap server is not a useful solution for pay-per-use systems.

4.2 Delegated CoAP Authentication and Authorization Framework (DCAF)

In addition to papers mentioned above, some interesting solutions are proposed to Internet Engineering Task Force organization (IETF). One interesting approach is Delegated CoAP Authentication and Authorization Framework (DCAF) [16]. This draft addresses authentication, secure communications, and authorization problems for securing CoAP protocol.

DTLS protocol in PSK mode is used for authentication and secure communication. DCAF defined 4 agents:

- Client (C), may be constrained or not constrained device,
- Authorization Manager (AM), performs authorization request for client,
- Resource Server (RS), object that is supposed to be accessed
- Authorization Server (AS),

In general case client obtains authorization ticket in JSON from format authorization server using authorization manager for selected resource server. The ticket consists of 2 parts - Face and Verifier. Verifier contains the key for C-RS secure communication and is supposed to be used only by client. Face in its turn contains authorization information and may contain C-RS key. If face contains key it is supposed to be encrypted with common key K_AS-RS.

Secured connection between client and RS is installed in PSK mode. Face part of the ticket is provided to RS in psk_identity field as JSON string representation or base64 encoding of encrypted object. RS either decrypt psk_identity or uses HMAC to generate a key.

The protocol described above is very well detailed. At the same time the procedure of access token grant looks controversial (that may happen in protocol draft). The ticket is requested for one resource and one or more actions on that resource but is granted in form of some abstract role and relations between resource, actions and a role are not specified.

The main problem of the protocol is that it grants an access to only one resource on RS. Hence, if client has an access to more than one resource then it is supposed either to maintain a few sessions to rehandshake every time it needs to access new resource (which can be expensive and time consuming).

4.3 Access Control Framework for Constrained Environments

One more interesting protocol that can be found in IETF drafts is Access Control Framework for Constrained Environments [17]. As well as DCAF, this approach is proposed for CoAP and DTLS protocols. Protocol draft defines 2 levels of access control:

- Protocol authorization where server makes a decision if it should run a requested protocol with certain client.
- Resource authorization where server allows or denies performing an operation on the requested resource for certain client.

Protocol authorization does not require resource authorization but resource authorization requires some protocol authorization.

Protocol identifies 5 agents in the system:

- Original Client (OC) emphasizing that real client may be hidden behind the middleware.
- Authorization Manager (AM) that requests access token.
- Authorization Server (AM) that is in charge for token request verification and token issuing.
- Resource Server (RS) constrained device that contains resources.
- Resource Owner that specifies the access policy for AS.

The OC using the AM obtains access token that includes CoAP resource identifier, CoAP action, OC identifier, AS identifier, additional conditions that specify if token can be accepted. In addition to information above token may contain sequence number that is maintained per RS to protect token from replay attack. The token is provided in JSON format and is supposed to be signed using JSON Web Signature (JWS) [18] or encrypted with JSON Web Encryption [19] (JWE) if it is send over an insecure channel.

The OC can transfer the token to the RS either using TLS Authorization Extension [20] or using specific CoAP resource. Resource URI should be defined as './well-known/core/authz/' or using a specific location on the RS, supplied by the AS. The token is verified upon receiving. During verification process RS checks if token was revoked, if issuer is reliable and if signature is correct.

The authorization process occurs upon receiving the request. RS verifies that token is not expired, that token is bound to the requested subject and that token is authorized to perform the operation on resource (including action and local condition).

4.4. Summary

The review of the above solutions shows in spite of the fact that many researches are working on the problem the solid and reliable solution does not still exists. Some of the researchers like [12], [13], [14] develop interesting but not detailed and evaluated solutions. The solution proposed by OMA is very detailed but too computationally complicated to be implemented for a sensor node. DCAF and [17] may be defined for resource control level only and cannot be applied in the case when only device access level need to be applied.

5 Solution overview

One possible solution for the problem stated in Chapter 3 is described in [21]. Authors of this draft propose 2 dynamic access control modes for applications that are based on CoAP and DTLS protocols.

- Derived key (DK) mode is based on PSK mode and provides dynamic access based on symmetric key cryptography.
- Authorized public key mode is based on RPK mode and involves asymmetric cryptography computations.

Both suggested modes are designed for device level access control (client that has access to the device can access any resource on the device). Since the goal of the thesis is to find a solution for the problem that requires minimum resources than derived key mode is more promising and will be described in details and evaluated.

In this solution 3 types of agents are identified.

- Resource Server (RS) is an object (constrained device) that hosts CoAP resources.
- Client (C) is a subject (device) that connects to Resource Server in order to access one or more resources.
- Trusted Anchor (TA) is server that has trust relations with Resource Servers and at the same time keeps access control policy that regulates clients' access to resource servers.

5.1 General workflow

The derived key mode resembles the PSK mode with the difference that keys are not explicitly provided to the RS. Instead RS and TA share the common secret key ($K_{RS\text{-}TA}$) that is not known to any third party. The workflow for DK mode is shown on the figure 5.1.

In the initial step client has to request an access token from the TA. The access token consists of 2 parts: a key that will be used between the client and the RS ($K_{RS\text{-}C}$) that is a binary value of 128 or 256 bits, and a nonce. The access token is supposed to be sent over a secure channel in spite it is not mentioned in the protocol.

After receiving an access token the client can initialize a DTLS connection with RS. The connection is initialized in the PSK mode. The $K_{RS\text{-}C}$ key is kept in secret and used as PSK while the nonce is sent to the RS in psk_identity field of ClientKeyExchange message. Upon receiving the nonce, RS can generate $K_{RS\text{-}C}$ from nonce and $K_{RS\text{-}TA}$ and successfully complete a handshake.

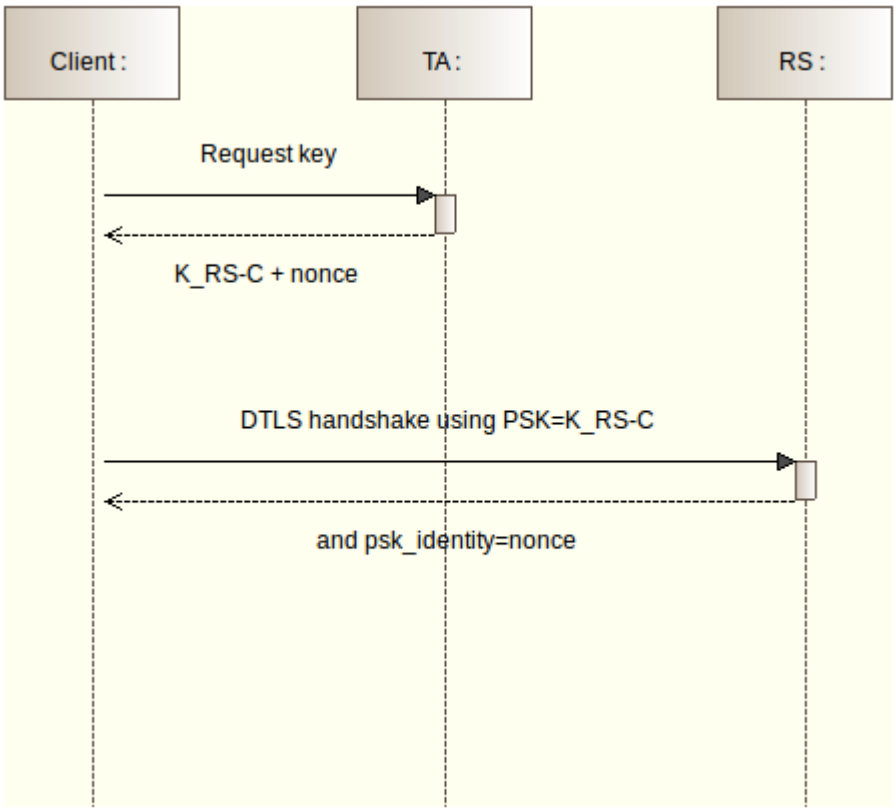

Figure 5.1.

Protocol draft [8] does not define precise format of the access token. Also, authorization and authentication of the client by TA is not in the scope of the document.

5.2 Nonce content and format

Nonce contains necessary information that RS needs to authorize the connection: client identifier, identifier of TA that issued the token, sequence number of nonce per RS (this sequence number is incremented every time a new nonce is generated for certain RS). As psk_identity field in ClientKeyExchange have to be a UTF-8 string then nonce have to be sent in a string format. In the protocol draft the following parts are defined:

- The constant nonce prefix “DK” which implies to RS that derived key mode is supposed to be used.
- TA identifier that is represented in string format or converted to string format.
- Client identifier that is represented in string format or converted to string format.
- Sequence number that is 32 bit integer converted to string format.

Parts of the nonce mentioned above are concatenated in sequence they are mentioned using a splitter symbol “.”. Example of the nonce is shown on the figure 5.2.

DK.TA_1.Client_1.5

Figure 5.2. Nonce Example

5.3 Calculation of the key

Both TA and RS derive the K_{RS-C} key using the nonce and the K_{RS-TA} key based on HMAC that utilizes SHA256 hash function [22]. The key is calculated using the equation 5.1 where K_{RS-TA} is HMAC secret and nonce is HMAC data.

$$K_{RS-C} = HMAC(K_{RS-TA}, nonce) \quad \text{.....(5.1)}$$

HMAC produces the 256 bit value that can be used completely or truncated to the most significant 128 bits.

5.4 Key expiration and anti-replay protection

In order to protect the nonce from reuse and implement a key expiration authors offer a to use sliding window based on the sequence number. The RS must maintain the list of minimum 32 (recommended 64) recent sequence numbers where maximum number is the biggest sequence number that was used. Each time the RS receives the nonce it has to validate if this number is bigger than the minimum number in the list, marked as unused in sliding window or bigger than maximum number that was used. So if the sequence number value is less than smallest value in the list than the key is considered to be expired and if the number is marked as used the RS assumes that this is a replay attack.

5.5 Key revocation

If TA needs to revoke the key that was not expired or terminate existing session it sends a request to RS with sequence number of a key that is supposed to be revoked. Upon request RS marks the certain sequence number in sliding window as used and terminates the active DTLS session if any.

5.5 Theoretical analysis

The protocol described above is a solid and well defined concept. The first important advantage of this protocol is a narrow application area. Unlike DCAF this protocol utilizes DTLS authentication to control access at the device level where all resources hosted on the RS are available to any client that has a right to connect. That makes it possible to access different resources without re-handshake or keeping additional sessions alive.

It is worth noting that this protocol neither make/require any changes in the DTLS standard nor in the CoAP standard draft. In practice that means that no modifications in DTLS or CoAP implementation libraries required and the protocol may be implemented on top of already existing code.

At the same time suggested protocol has a few drawbacks. First, this protocol does not offer any solution for resource level access control. But that can be considered as minor disadvantage because this framework can be built separately or on top of the existing protocol.

The second important concern is content and a format of the nonce. In addition to the existing data (TA identity, client identity, and sequence number) it may be good to add the key size (as it was mentioned above it can be 128 or 256 bits) to improve flexibility. The format of the nonce can become a problem when it comes to fields separation with special symbol "." and identifying derived key mode by sequence of 2 symbols at the beginning of the nonce. Defining a special symbol (or a sequence of symbols) as a separator may introduce bugs in implementation and settings parts. Mode recognition by sequence of symbols may lead to mistakes when the key in the PSK mode is defined starting with the same sequence of bytes.

In order to improve the protocol the nonce could be redefined in form of binary structure (Table 5.2). The total size of the structure is 37 bytes. The first 3 bytes are the constant sequence 0x0C, 0x44, 0x4A that identifies the DK mode. The fourth byte is the identifier of the Trusted Anchor. Next sequence of 12 bytes is a client identifier is followed by 12 bytes of resource server identifier. Key size defines the size of the key that is supposed to be used 0 stands for 128 bits and 1 for 256 bits. The last 8 bytes of the structure is a sequence number of the key.

Table 5.2. Structure of the binary nonce

Field	Size (bytes)	Comment
dk_id	3	Identifier of the derived key mode.
ta_id	1	Identity of the TA as number
client_id	12	Identity of the client as number
rs_id	12	Identity of resource server
key_size	1	size of the key to generate
sequence_num	8	64 bit sequence number

The purpose of RS identifier is additional protection in case to resource servers have the same key $K_{RS\text{-}TA}$. In this situation if the RS identifier is not present in the nonce a client can generate a nonce and key for one device and then use it to access another device.

Unfortunately it is not possible to send a binary structure in the psk_identity field of the ClientKeyExchange message according to TLS standard specification. We see 2 ways to overcome this problem.

The first possible solution for the first problem is to define the extension using standard TLS extensions filed in the ClientHello message. In this case the ClientKeyExchange message may contain a client string name in the psk_identity and binary structure defined above can be sent using the extension_data field of the ClientHello. The main disadvantage of this method is that it requires the additional TLS extension definition, that cause the changes in DTLS implementation both on client and on all RS.

One more possible solution is to represent a binary structure as a string to send it in the ClientKeyexchange message. That can be achieved by using some standard encoding algorithm that can convert binary representation to string like base64 [23].

The approach that utilized base64 encoding seems to be more promising than one that defines additional DTLS extension because it does not require any CoAP or DTLS code modifications of a well-designed library. In general case, it cannot be predicted how the key could be obtained so designers of the DTLS library have to define a way to extend key retrieval without internal code modification. Hence, it can emphasize that the DK mode is suitable even for proprietary libraries that doesn't provide access to source code.

The protocol draft contains only general description of the key revocation mechanism. We can improve the protocol by defining this part in more details. The request for key revocation does have to contain the TA identifier and RS identifier together with sequence number of a key that have to be revoked. This message can be send using DTLS session in pure PSK mode but the DTLS handshake has too much overhead. Moreover the constrained device has limited slots for DTLS sessions so the handshake will not be always possible.

Since the information that is contained in the key revocation request is not confidential but has to be authenticated we suggest avoiding establishing of a secure connection and protect the message with Message Authentication Code (MAC) based on the HMAC algorithm. The MAC can be computed with key $K_{RS\text{-}TA}$ so it can be easily verified on a constrained device.

The structure of the key revocation request is shown in the table 5.3. It starts with 1 byte of the TA identifier followed by 12 bytes of the RS identifier and 8 bytes sequence number. The last 32 bytes is a message authentication code computed based on information in the structure with key $K_{RS\text{-}TA}$.

Table 5.3. Structure of the key revocation request.

Field	Size (bytes)	Comment

ta_id	1	Identity of the TA as number
rs_id	12	Identity of resource server
sequence_num	8	64 bit sequence number
MAC	32	message authentication code

The key revocation request can be send over any standard or custom application level protocol. If CoAP protocol is supposed to be used on the device then specific endpoint should be defined for key revocation and message should be sent as a payload of ACK request.

The solution that is defined above including refinement for the nonce and key revocation will be implemented and evaluated further.

6 Implementation details

In order to verify and evaluate the protocol 3 agents that were defined in the previous chapter have to be implemented.

- The client is implemented in Java as a desktop application.
- The Trusted Anchor (TA) is implemented in Java as a web service application.
- The Resource Server (RS) is implemented in C on a constrained hardware platform.

Deployment of applications is shown on the figure 6.1.

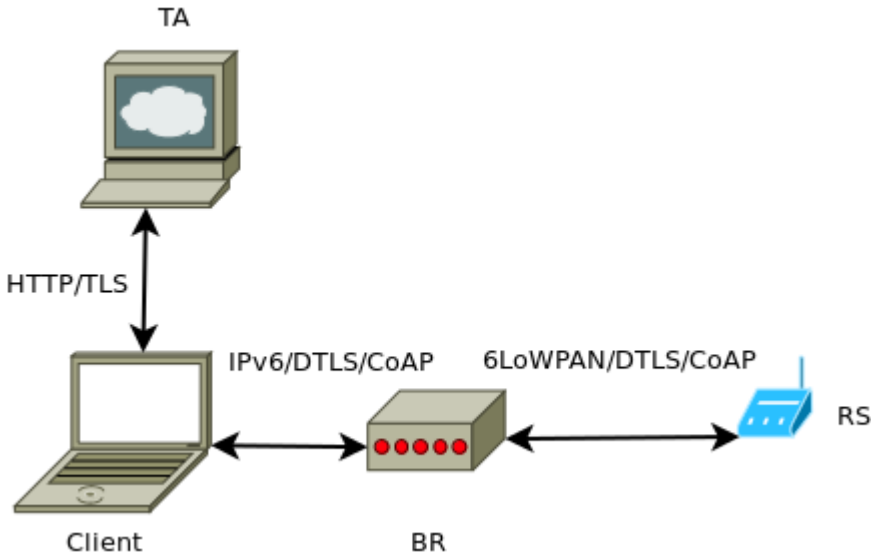

Figure 6.1 Deployment of applications.

TA is running on a web server. The client is running on a desktop computer or a laptop and sends HTTP requests to the TA over a secure internet connection. The client is also connected to the Border Router (BR) over the serial port utilizing Serial Line Internet Protocol (SLIP). The RS, in its turn, is connected to the BR over 802.15.4 low power radio.

6.1 Resource server implementation

The resource server development is includes not only the design and implementation of a solution that was described in Chapter 5 but also the selection of the hardware platform, the operating system (OS) and necessary libraries.

6.1.2 Hardware platform

The resource server is supposed to be implemented on the hardware platform that has limited resources so it can be considered a constrained node. But at

the same time it should provide enough memory and computational power to run the operating system, IP stack, DTLS, CoAP protocol as well as access control implementation and test application. The 16 bit processors can address up to 64K of memory that is probably not enough to keep all this components. Hence, 32 bit hardware platforms need to be used. One more important requirement for the platform is presence of the 802.15.4 radio unit. 2 platforms were found that meet requirements above ST32W [24] and CC2538 [25]. Both of these platforms are based on 32 bit ARM-Cortex M3 core, have the built in 802.15.4 radio unit and the same about of flash and Random Access Memory (RAM). The main benefit of the CC2538 is presence of the more advanced accelerator of cryptographic operations that supports SHA2 and AES-128/256 while ST32W supports only AES-128 hardware acceleration. One more important advantage of the CC2638 is a hardware pseudo random number generator that can be initialized with random number generated by radio module. Due to the benefits above the CC2538 was chosen as hardware platform.

The development kit CC2538DK [26] that is based on CC2538 was used for implementation and evaluation. The development board has following relevant characteristics.

- processor - CC2538F512RKU
- flash - 512Kb
- RAM - 32Kb
- clock frequency - 32MHz,
- the voltage - 2.1V
- the processor core current - 13 mA
- oscillator frequency - 32.768kHz.

Moreover, the CC2538DK development board has 4 built in LEDs, accelerometer and a light sensor [27] that helps to build a test application for protocol evaluation.

6.1.1 Operating system

The OS is necessary in order to simplify the development of the test bed and avoid custom implementation of different services (like the IP stack). At the same time operating system is supposed to have small code overhead and good power efficiency. Two operating systems were considered as possible options: TinyOS [24] and Contiki OS [25].

For this project the Contiki OS version 2.7 was used. The Contiki OS is the open source, well maintained operation system that is designed especially for the IoT [26]. One of the important benefits of this operating system is a support for the communication IP protocol stack. It includes IPv6 (with 6LoWPAN), TCP and UDP protocols. The Contiki OS provides option to switch off unused units of the communication stack. For our implementation we do not include TCP and leave only IPv6 (with 6LoWPAN) and UDP protocols.

One more advantage of the Contiki OS is the implementation of the simple multithreading concept called protothreads that from the one side simplifies implementation and from the other side has small code overhead and is not demanding to computational resources [27].

Moreover the Contiki OS also implements various Radio Duty Cycling (RDC) protocols. Devices that utilize RDC keep the radio off most of the time and waking up in periods (constant or variable) to check if there any nodes that are ready for communication. As a radio unit is the most power consuming part of the constrained device using RDC can be a great energy saving option. The ContikiOS supports XMAC, CXMAC and its own ContikiMAC protocols. As the ContikiMAC provides the best energy saving characteristics (it keeps the radio off 99% of the time [28]) this RDC protocol was chosen for the test bed implementation.

In addition to this, the Contiki OS comes with built in CoAP implementation Erbium [29]. It supports both server and client modes and has a small code overhead. For the used version of operating system Erbium implemented the version 13 of CoAP protocol draft that has only minor differences with the latest version 17 that was available for the moment of this thesis.

One more important benefit is that the Contiki OS provides built in JSON support that might be helpful for evaluation of possible solutions that involve access tokens in JSON format.

6.1.2 DTLS library

The DTLS implementation that can be used in a test bed is supposed to fit following requirements. At first, as a DTLS library has to run on the constrained hardware platform it should be implemented in C and it should be an open source solution so it can be easily optimized. The second important requirement is that DTLS implementation has to support PSK mode and in the best case TLS_PSK_AES_128_CCM_8 ciphersuite. From a variety of existing implementations GnuTLS [30], MatrixSSL [31] and tinyDTLS [32] were considered as possible options. Finally, the tinyDTLS library was chosen because it not only meets all the requirements mentioned above but also has Contiki OS support and is optimized in size and performance for constrained devices.

At the same time a few modifications had to be implemented.

- The code for TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 and Elliptic Curve Cryptography (ECC) support was disabled for compilation to reduce the code footprint.
- The software base pseudo random number generation (PRGN) was replaced with hardware accelerator for PRNG generation.

- Functions that utilize the SHA2 hardware accelerator for HMAC calculation were implemented to replace the software based SHA256 calculation.
- DTLS session handler was extended to keep a pointer for custom information and search of existing session by custom information.
- The new event DTLS_EVENT_CLOSED was declared. This event is send when the DTLS session is about to be closed but session information was not removed yet. It can be handled by the event function together with DTLS_EVENT_CONNECTED.
- The function that destroys the DTLS session and can be called outside of the tinyDTLS module was implemented as a part of DTLS.

Modifications and extensions mentioned above are not application specific and have to be supported by the DTLS implementation so we treat this code as part of the DTLS library and not our application.

6.1.3 DTLS and CoAP integration

In order to develop a test bed, the tinyDTLS library was integrated in the Erbium application. DTLS support was implemented as an option and can be disabled or enabled in compile time. The following modifications were made in Erbium CoAP implementation.

At first support for 2 UDP connections was added to Erbium. One connection is supposed to be secured with DTLS while another handles insecure requests. In order to prevent insecure CoAP calls to resources that are supposed to be protected the additional resource flag IS_SECURE was declared and can be used to prevent resource from insecure access.

Callback methods of the tinyDTLS library get_from_peer and send_to_peer, which are used to process data from upper level protocol and send data with lower level protocol respectively, were implemented in Erbium. The get_from_peer delegates requests to coap_receive method that is a part of Erbium and implements CoAP message processing. The send_to_peer method, in its turn, sends data over the UDP protocol.

As the Erbium implementation can have only one instance per processor one dtls_context global variable was added, that handles pointers to callback methods. It was initialized with get_from_peer and send_to_peer methods mentioned above and get_key method, provided from the test application, that returns a key depending on PSK identity.

The coap_receiver thread is responsible for receiving data from lower level protocol and delegating processing to coap_receive method. In order to implement DTLS integration data processing was delegated to the dtls_handle_message method that is a part of tinyDTLS library. At the same time in coap_send_message the code for sending data over UDP protocol was replaced with call of dtls_write method. The general structure is shown on Figure 6.2.

The coap_receiver process receives the new packet and checks by destination port number if it has to be processed as secure or insecure. If the data was sent to insecure CoAP connection the request is processed as any other CoAP request by coap_receive method. The only difference from initial protocol implementation is the additional check of IS_SECURE flag of target resource. If the data was sent to DTLS connection dtls_handle_message method is called for further processing. The dtls_handle_message in its turn checks if the message belongs to DTLS protocol. Then if the message belongs to the record protocol it is decoded and processed with the coap_receive method. In case a response is supposed to be send that response is encoded and sent back to a client. If the received message is a handshake or alert message it is processed with the logic of the DTLS protocol.

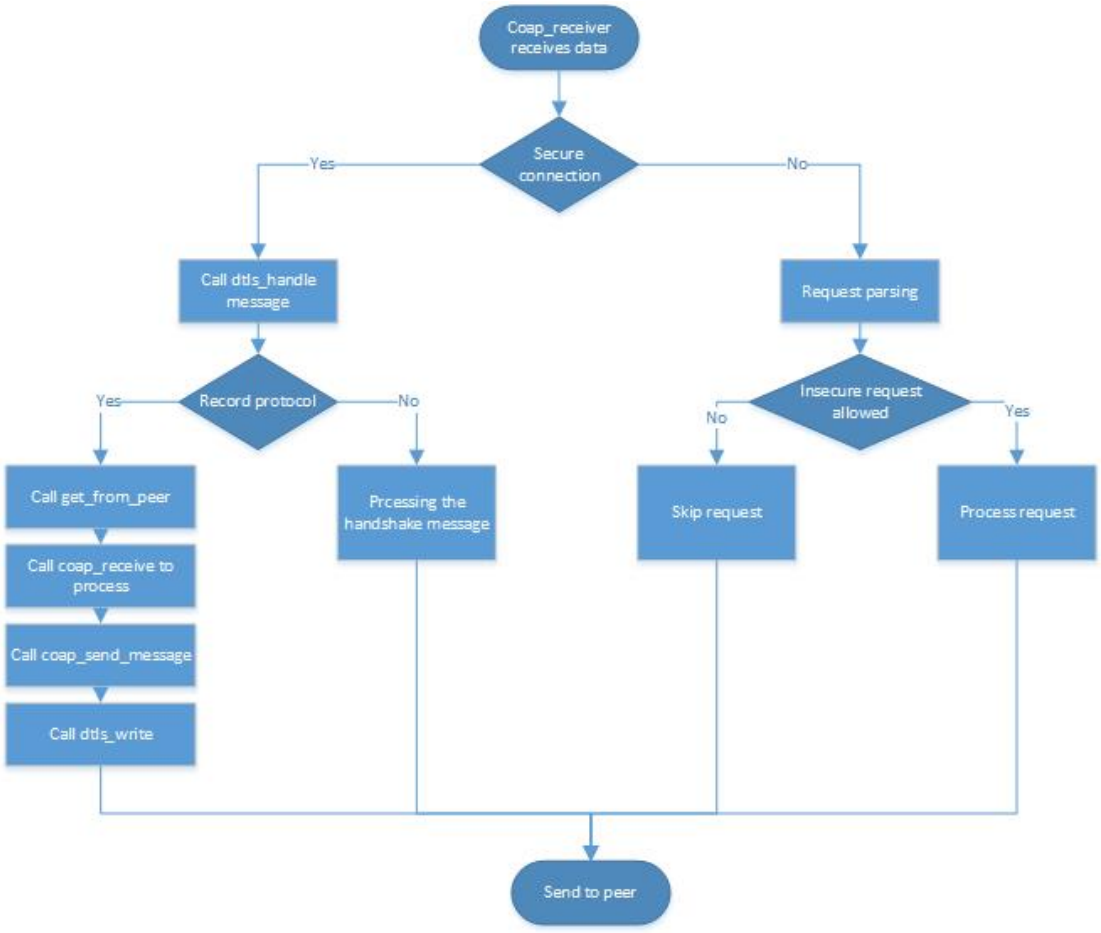

Figure 6.2. Message processing flowchart.

6.1.4 Key derivation

The key derivation logic implemented as the function that accepts the PSK identity and length of the PSK identity in bytes and returns the status of the operation (0 if key is generated and non 0 if an error occured), the key, the length of the key in bytes, the TA identifier and the sequence number from the nonce. The key derivation flowchart is shown on the Figure 6.3.

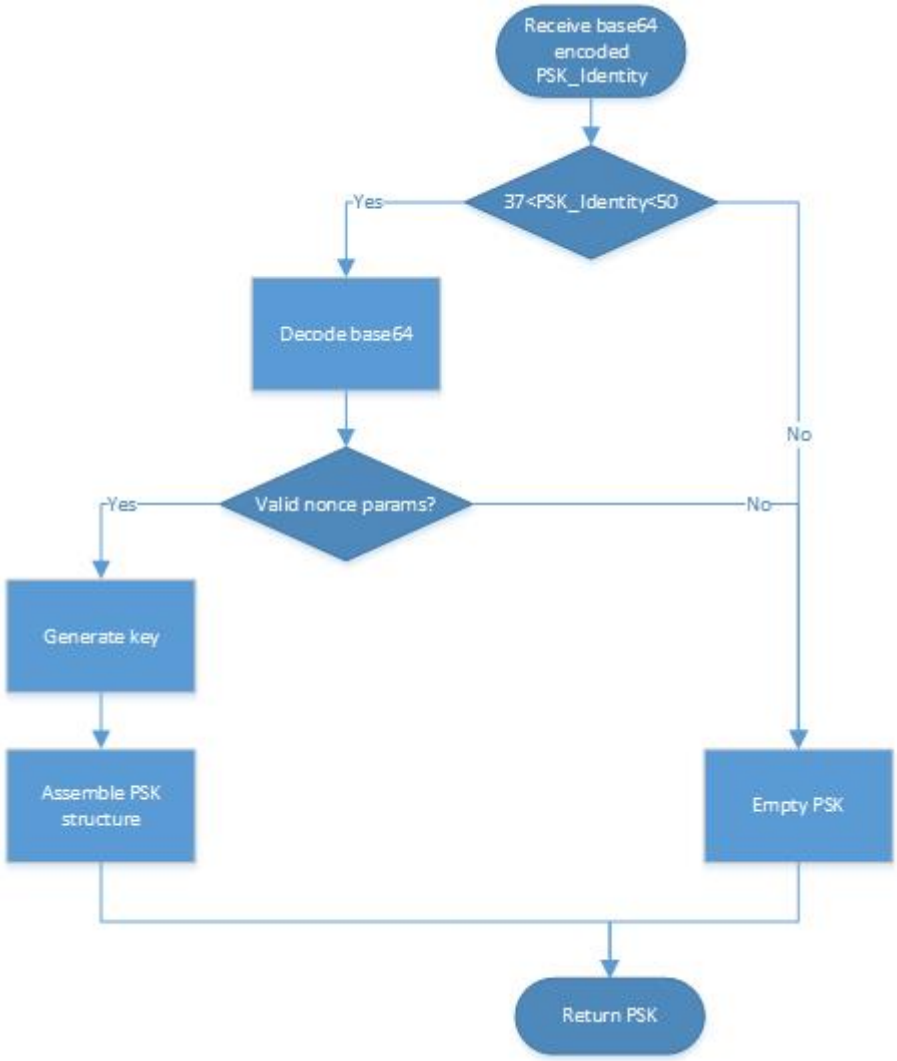

Figure 6.3. Key derivation flowchart.

In order to prevent redundant base64 decoding, the function verifies that the length of the PSK identity field is at least 37 (the size of a nonce) but does not exceed 60 bytes (the maximum length of nonce Base64 encoding). If the length of PSK identity is within the acceptable limit the nonce is decoded from Base64. If the result of the decoding is exactly 37 bytes than data of the nonce is verified in the following order.

- First 3 bytes of the nonce identify the key mode and are equal to 0x0C, 0x44, 0x4A
- RS identifier specified in the nonce is equal to the RS identifier of the device.
- The TA identity is known to the device and sequence number of the nonce satisfies requirements of the sliding window of the specified TA.

If all checks of the nonce are successful then the derived key is generated and truncated if it is required by the key size parameter of the nonce. As the implemented access control protocol is designed to be used together with the

DTLS protocol the key generation function utilize the HMAC calculation module of the tinyDTLS library to reduce the code footprint.

6.1.5 Sliding window

The sliding window has a size of 64 values as recommended by the protocol specification. It is implemented as a structure and contains 2 integer variables (window and sequence_number). Each variable is 64 bit size. The sequence_number variable contains the smallest sequence number known by the window. The window variable, in its turn, represents the status of used keys for the sequence of next 64 numbers. Each bit can be either 0 what means that the key was not used or 1 what means that the key was already used. Bits are numbered from the less significant bit to the most significant bit so the bit number 0 represents the status of the key with the sequence number equal to sequence_number field. The most significant bit represents the status for the key with the number sequence_number+63.

The verification of a nonce with the sliding window is implemented in a following way.

- If the sequence number provided with the nonce is less than the sequence_number value of the sliding window then the nonce have to be ignored.
- If the sequence number provided with the nonce is greater than sequence_number + 63 than nonce can be accepted.
- If the sequence number provided with the nonce is between the sequence_number and sequence_number + 63 then nonce can be accepted if respective bit in the window variable is set to 0.

The key is marked as used in the sliding window either when the DTLS handshake is complete or when the key revocation request is received. If the sequence number provided with the nonce is greater than sequence_number + 63 than the window variable is set to 0 and the sequence_number variable is updated with the new sequence number. Otherwise if a sequence number is in between of the sequence_number value and sequence_number + 63 than the respective bit of the window variable is set to 1. If the sequence_number variable of the sliding window is greater than the sequence number of a key that have to be marked as used then the sliding window is not updated.

6.1.6 Key revocation

The key revocation is implemented as a CoAP resource that is not protected with transport security. The resource utilizes the DELETE method of the CoAP protocol and the request is sent as a row sequence of bytes in the payload. The request is not encrypted but protected with the MAC. As HMAC calculation may cost significant computational resources, following conditions are verified when the new key revocation request is received.

- The RS identity specified in the request has to be equal the resource identity of the device.
- Either the DTLS session for the sequence number specified in the key revocation request exists or key is marked as unused in sliding window.

If verification above was successful the MAC is calculated with key $K_{RS\text{-}TA}$ and compared with the received MAC. In case if the message sender identity is not confirmed the request is ignored. Otherwise the sliding window is updated and the DTLS session is removed if it exists.

6.1.6 Test application

The test application is implemented as a CoAP resource that is protected with the DTLS protocol and performs a simple function of switching Light-Emitting Diod (LED) on the evaluation board.

In addition to this it defines callback functions for the tinyDTLS context and on_event function that handles tinyDTLS events. This function updates sliding window information when the DTLS handshake is finished and session is marked as connected. It also releases memory for custom information that is sticked to the session when the session is about to be closed.

The get_psk_key callback function is called by the tinyDTLS when it needs a PSK for the DTLS handshake. The implemented handler supports both pure PSK mode and derived key mode.

The test application also contains service code to configure the DTLS context and start CoAP server.

6.2 Border router

The border router is the CC2538DK board that is programmed with RPL border router application. That application can be found in Contiki OS examples. The router is connected to PC with the Serial Line Internet protocol (SLIP). In order to run the SLIP protocol the tunslip6 tool from Contiki 2.7 distribution was used. This tool creates the additional network interface with the specified prefix and allows IP packet exchange between the PC and the RPL border router.

6.3 Trusted anchor implementation

The Trusted Anchor is implemented in Java as a web application that provides the REST service with 2 endpoints.

The key generation endpoint (token/generate) creates the new nonce and derived key pairs. As input it accepts the JSON object that contains the alias of the client that sends the request and the alias of the resource server that needs

to be accessed. The TA finds existing RS and client identifiers by aliases in the database, generates the nonce and the key and increments the sequence number counter. The response is a string representation of the JSON object that contains nonce encoded as Base64 and a session key K_{RS-C}.

The key revocation endpoint (token/revoke) is designed to send key revocation requests. As input it accepts the JSON object that contains the alias of the resource server were the key have to be revoked and the sequence number of that key. The TA finds the the RS identifier by the alias and sends key revocation request to the selected RS.

The REST service is secured with the TLS/SSL protocol based on generated X.509 certificates. Since our main goal is to evaluate the part of the protocol that have to run on the constrained devices we did not implement any authentication.

6.4 Client implementation

The client application is an application implemented in Java. It is designed to execute 3 main functions:

- Obtain the nonce and the derived key K_{RS-C} from the TA.
- Perform the DTLS handshake with the RS using granted nonce.
- Send the CoAP request over the DTLS record protocol to RS.

The nonce and the derived key are obtained in a JSON object from the key generation endpoint. In order to implement the DTLS handshake and the secure CoAP communication we used Scandium open source DTLS implementation and Californium CoAP implementation. These frameworks were selected because they provide all necessary functionality and are easy to integrate.

7 Evaluation

The implemented resource server part of the access control framework was evaluated on in terms of code size, memory usage, computational time and energy consumption. Since the nonce is transmitted in the psk_identity field of the ClientKeyExchange message the same as for normal PSK mode we assume zero transmission overhead for the key derivation mode.

The evaluation was performed separately for key derivation and key revocation requests since those parts of the protocol are low coherent and can be redefined separately of each other.

7.1 Evaluation methodology

In order to evaluate code size and memory usage the arm-none-eabi-size utility was used. For more detailed information about RAM and ROM usage we used the tool arm-none-eabi-objdump. Both tools are included in the GNU toolchain for ARM processor utility. To perform a measurement for a certain functional part of the code that is in charge of specific functionality, it is removed from compilation with the #define C directive and the value is calculated as delta of the application size with and without the functional part.

The computational time is measured with the contiki energest module. This module is based on the real time clock and measures usage time of different platform units separately (CPU time, CPU time in low power mode, radio listening time and radio transmission time). The real time clock on CC2538EM platform works on 32.786kHz frequency that allows to perform measurements with resolution 0.03 ms. The energest module accumulates the number of real time clock ticks in 64 bit unsigned values since its activation. That capacity is more than enough to measure the computational time of the most of access control operations.

Energy consumption is calculated from time that is measured with the energest module according to the equation 7.1.

$$E=U*I*t \quad (7.1)$$

In this equation U is a power supply voltage taken from platform documentation, I is an average current for the respective module from processor specification and t is the time value measured with the energest. The energy value is calculated for each model and then summed up to take into account not only energy used by processor for computation, but also potentially lost radio duty cycles.

In order to evaluate how many requests can be send by an attacker per second we measured the Round Trip Time (RTT) between client and server with a ping request. As a ping request processing time is negligible and the network consists of two nodes so the route is deterministic we assume the One Way Delay (OWD) is a half of RTT. Hence, the number of requests that can be sent

by an attacker during the period of time is the length of the period divided by OWD.

The measurements were performed with ContikiMAC radio duty cycle protocol.Java client running on the PC was sending either a DTLS handshake requests sequence or a key revocation request the RS running on the embedded platform depending on the evaluated function. Data was collected 30 times for each function.

The collected data were analysed to make a conclusion about performance of the access control framework and its resistance for Denial of Service (DoS) and Drain Battery Attack that are specific for constrained devices.

7.2 Access control framework evaluation

The total size of the access control framework is 1708 bytes including 1636 bytes of code, 48 bytes of static information and 24 bytes allocated for global variables. This number also includes the 16 bytes long RS identifier and 78 bytes long key $K_{RS\text{-}TA}$. The part of the key revocation functionality is 392 bytes in total.

The performance of the framework was measured separately for the key derivation function and for the key revocation function. Measurements results are presented in the table 7.1.

Table 7.1. Performance measurements of the key derivation and revocation.

	Computation time (ms)	Energy (μJ)
Key derivation	2.30	62.79
Key derivation with SHA2 accelerator	0.45	12.29
Key revocation	2.25	61.43
Key revocation with SHA2 accelerator	0.35	9.56

The key derivation with software SHA256 computation takes in average 75.47 ticks or 2.30 milliseconds while the same computation with SHA2 hardware accelerator takes 14.6 ticks or 0.45 milliseconds. The key revocation with and without SHA2 accelerator takes 73.7 ticks (2.25 ms) and 11.4 ticks (0.35 ms) respectively.

7.3 DTLS evaluation

In order to compare the impact of the key derivation mode and SHA2 accelerator on the DTLS handshake we measured the processing time for each message send by client. This time is measured starting from receiving the message and until the reply is sent. Moreover the total handshake time starting from the first ClientHello message and ending with processing of the last Finished message was measured to get an idea about the maximum number of handshakes that can be processed by the device within a unit of time.

The total size of the tinyDTLS library (including the access control framework) is 21592 bytes. This code footprint includes 19368 bytes of program, 140 bytes of constant data and 2084 bytes allocated for global variables.

Results of performance measurements are presented in the table 7.2

Table 7.2.

	Time (ms)	Energy (μJ)	Time (ms) with SHA2 accelerator	Energy (μJ) with SHA2 accelerator
First ClientHello	2.32	63.29	0.72	19.80
ClientHello with cookie	3.55	96.78	1.32	35.99
ClientKeyExchange	0.37	10.16	0.06	1.66
ChangeCipherSpec	17.89	488.46	3.30	90.00
Finished	9.96	271.99	2.65	72.40
Total	38.64	1054.96	8.85	241.52

The minimum time for the complete handshake for SHA256 software computation is 533.05 ms and average time is 775.05 ms. In case the SHA2 hardware accelerator is used the minimum and average time is 511.65 ms and 711.11 ms respectively.

7.4 One way delay

The average RTT value was 362.95 ms. Hence, the average OWD value is 181.48 ms and we can assume that attacker can send 8.44 requests per second.

7.5 Discussion

Analysis of the access control framework code footprint shows that key derivation and revocation functions take 7.9% of the total DTLS implementation. The key derivation function takes 6.0% of total computational time per handshake. SHA2 hardware acceleration has a significant impact on the key derivation and revocation processing time for

DTLS handshake messages. The accelerator speeds up the key derivation computation 5.11 times and the key revocation computation 6.43 times. Also the accelerator speeds up overall computation time of the handshake 4.37 times.

We found out, by comparing overall handshake time with and without SHA2 hardware acceleration, that optimization of the computational time has almost no impact on total handshake time. Hence, the key derivation has no significant impact on total DTLS handshake time.

7.5.1 Attack analysis

In order to analyze the access control framework for drain battery and DoS attacks we assume following conditions.

- The constrained device uses the CR2032 coin battery with charge capacity 225 mAh [33]
- The attacker can send 8.44 requests per second

Battery drain attack can be performed either by sending the sequence of key revocation requests with well-structured valid data but wrong MAC or by series of attempts to perform a DTLS handshake with well-structured and acceptable nonce.

In case an attacker tries to send key revocation requests we assume that 8.44 requests per second can be transmitted. In this case the processor spends 0.07 μAh (2.25 ms * 16 * 13 mA) of battery charge per second. In other words, it requires 38 days to drain the battery with this method, which means that this type of attack is almost unfeasible.

In order to analyze the feasibility of the battery drain attack we assume the worst case that an attacker can perform a handshake in minimal handshake time (once in 0.5ms). The access control protocol requires the full DTLS handshake to be performed to authentify the client so we assume that requires 38.64 ms to reject the client with valid the nonce and the invalid key. In this case the processor spends 0.28 μAh (38.64ms * 2 * 13 mA) of battery charge per second. Or in other words it requires 9.3 days of continuous sending to drain the battery that makes this attack unfeasible.

A denial of service attack and be considered feasible and successful if an attacker can send a request to the resource server that makes it unresponsive. An attacker can perform this either by sending requests that require significant time to be processed or requests that block shared resources. As with battery drain attack we analyzed the if DoS attack is feasible either by sending key revocation requests or by performing a handshake.

Processing of each key revocation request without SHA2 accelerator takes 2.3 ms. Hence, the processor can service 434.7 requests per second that is much more that 8.44 requests per second that can be serviced by the network.

In DTLS handshake ChangeCipherSpec message requires the longest processing time of 17.89 ms. Hence, the processor can service 55.90 requests per second. That is more than amount of requests that can be transmitted over the network. So the DoS attack cannot be performed on the DTLS handshake.

At the same time the problem comes from the DTLS protocol specification. The DTLS state machine is initialized on the Resource Server right after the ClientHello message with valid cookie is received. Since a constrained device can handle a very limited number of slots for DTLS sessions (in some cases the device can handle only 1 session) the potential attacker may perform a Denial of Service (DoS) attack by sending ClientHello messages and keeping all session slots busy.

7.5.2 Protocol improvements.

As was mentioned above that the protocol is well defined and doesn't performs worse than the original DTLS handshake in PSK mode. But at the same time it has important drawbacks that have to be solved.

Roles

The first disadvantage is that access is supposed to be granted to all resources of the device while the resource server can contain different resources that is supposed to be accessed by different users (for example configuration resources that have to be accessed only by the device vendor and resources that can be accessed by the user). The number of these resources is usually limited. For this case we can offer to add the definition of roles to the framework.

A set of roles that can access the resource is defined as a 64 bit integer value. In this case the number of a bit in the value identifies the role and the value of the bit means the access permission. If the bit is set to 1 if access permission is granted to the role and if it set to 0 then access is not granted. Each nonce, in its turn, contains additional a 64 bit integer that defines the set of roles that is assigned to the client. The same as for resources if user has a role the bit is set to 1 and to 0 otherwise. The set of roles is assigned to the DTLS session and verified for each request with bitwise '*and*' operation.

This approach has allows to define up to 64 different roles per resource server. That should be enough for a constrained device. It has only 8 bytes of communication overhead and a few instructions of computational overhead. Moreover this role base approach lets the client to access different resources without reestablishing DTLS handshake connection.

Bulk key revocation

In some cases it can be required by the Trusted Anchor to perform the key revocation procedure for multiple sequence numbers. The example of such case is an external request to TA to revoke all existing keys granted to the certain client on the certain resource server. TA has to send a set of revocation

requests to the specified device starting from the last generated key sequence number and continuing with all previous sequence numbers that meet the size of the sliding window. As the new MAC has to be generated for each request and the same data has to be send multiple times that procedure has significant communication and computation overhead.

In order to improve the key revocation procedure we offer to change the key revocation request format so it contains instead of one sequence number the list of sequence numbers from the latest issued to the device and up to 63 previous numbers if any for this device. The structure of new key revocation request is shown in the table 7.3.

Table 7.3. Structure of the key revocation request.

Field	Size (bytes)	Comment
ta_id	1	Identity of the TA as number
rs_id	12	Identity of resource server
sn_lenght	2	Length of sequence numbers array
sequence_nums	[1..255]	the array of 64 bit sequence numbers
MAC	32	message authentication code

In the table 7.3 ta_id, rs_id and MAC fields has the same meaning as before. The sn_lenght field is the number of sequence numbers sent within request and sequence_nums is the array of sequence numbers to be revoked. In this case we can reduce the computation time because all numbers are protected with same MAC so it is verified once and reduce the communication overhead as all numbers are sent within 1 request. The communication overhead in comparison with previous definition is negligible (2 bytes or 3.8% of the payload).

Key derivation procedure.

One more potential problem is the key derivation procedure. In case the client doesn't know the key or uses the wrong key the device performs unnecessary computations (key material computation, encrypting a Finished message and decrypting client's Finished message). This computational time can be significantly reduced if we change the nonce format and the key derivation procedure.

In order to perform early authentication the existing nonce is complemented with HMAC of the binary part of the nonce. The new structure of the nonce is defined in the table 7.4. The dk_id, ta_id, client_id, rs_id and sequence_num fields are the same as before. The mac_and_key_size is split in 2 parts: 4 MSBs can be 1 or 0 that means 256 bit or 128 bit MAC respectively and 4 LSBs can be 1 or 0 that means 256 bit or 128 bit generated key.

The MAC is generated by the TA with HMAC algorithm from the nonce and the key RS_TA and truncated if necessary. Then MAC is concatenated with the nonce and full nonce is encoded with Base64. The key C_RS is generated from the encoded nonce.

Table 5.2. Structure of the binary nonce.

Field	Size (bytes)	Comment
dk_id	3	Identifier of the derived key mode.
ta_id	1	Identity of the TA as number
client_id	12	Identity of the client as number
rs_id	12	Identity of resource server
mac_and_key_size	1	size of the key to generate
sequence_num	8	64 bit sequence number
Mac	[16..32]	MAC of the nonce

Upon receiving the nonce is decoded from Base64 and all necessary checks are performed. Then MAC of the nonce data computed and compared with the provided MAC. If verification fails no key is computed and the handshake is terminated.

In order to analyse this solution we can assume that HMAC computation is preliminary equal to the key revocation request processing time and equal to 2.25ms in average. At the same time nonce verification time is the same as key derivation time. Hence, the nonce verification, in case if the client knows the right key, requires additional 2.25ms or 5.78%. At the same time if the client uses wrong key the nonce verification saves more that 66.12% of computational efforts for wrong handshake.

DoS attack on DTLS session slots

As mentioned above there is a possible DoS attack on the DTLS protocol running on the constrained device with limited number of slots. In order to fix this approach with the access control framework it can be redefined as a DTLS extension. In this case the nonce information and verification data can be send in the extension_data field of the ClientHello. When the resource server receives the ClientHello message with valid server cookie and extension information the nonce is verified and state machine is create only in case if the nonce is valid. The nonce information have to be saved as a custom data for the DTLS handshake and used later when the ChangeCipherSpec message received to derive a session key.

In addition to preventing the DoS attack definition of the access control framework as the DTLS extension makes possible to early validate erroneous and expired nonces. Moreover, sending nonce as a binary value allows to reduce the code footprint by removing Base64 decoding function (it takes preliminary 300 bytes). At the same time the implementation of DTLS extension may require modification of DTLS protocol code that not always possible.

8 Conclusion

The intention of this thesis was to design and evaluate the access control protocol that is suitable for resource constrained devices that connect objects to the Internet of Things. We analyzed existing solutions and based our approach on the IETF draft that is based on the DTLS protocol in PSK mode. The draft was analyzed and a few improvements were offered before implementation and evaluation.

The protocol was implemented and evaluated on CC2538 platform that includes the low power ARM Cortex M core and 802.15.4 radio module. This hardware configuration is commonly used in IoT applications. Evaluation results shows that the access control framework increased computational effort of the DTLS handshake by 6.0%, increases the code footprint of the DTLS implementation by 7.9% and has no effect on the overall handshake time. We found out by analyzing computational time that the protocol is not vulnerable to Deny of Service or Battery Drain Attack.

Moreover, the DTLS handshake protocol in PSK mode was evaluated using the CC2538 platform and the ContikiMAC Radio Duty Cycle protocol. We found out the computational efforts required for processing handshake messages and average overall handshake time. We, also, compared results with software and hardware implementation of the SHA-2 hash function and found out that the hardware accelerator speeds up message processing computations in 4.37 times but has no effect on overall handshake time. Analysis of attack vulnerability shows that with a limited number of session slots that is expected in IoT applications it is easily possible to make the device unresponsive for about 2 minutes.

According to evaluation results we offered to add roles as the functional extension of the protocol. Moreover we suggest to improve the key derivation procedure to reduce computational efforts for processing fake or erroneous nonces and to improve the key revocation procedure to perform multiple key revocation in one request. In addition to this, we recommend to define access control protocol as a DTLS extension in order to prevent the Denial of Service Attack mentioned above.

The resulting protocol is feasible to use in the IoT. We recommend this approach for application that require dynamic centralized access allocation, reliable user authentication and authenticated encryption of data transmitted in both directions. It is possible to use both with CoAP as the application level protocol and any other application level protocol that is built on top of UDP transport. The typical usage example can be pay-per-use applications in the IoT.

8.1 Future work

In this thesis we focused on the scenario when one unconstrained device communicates (Client) with one constrained device (resource server). So it is desirable to complement the work with evaluation of the refined protocol for the case when both devices are constrained nodes. It is also interesting to evaluate the full time for the connection starting with sending the key generation request to the TA and up to the end of DTLS handshake. Since the protocol is designed to work only in single cast mode, after this evaluation we are planning to improve the protocol with multicast mode extension.

References

[1] IEEE Standard. IEEE 802.15.4-2003: Wireless medium access control and physical layer specifications for low-rate wireless personal area networks, ISBN 0-7381-3677-5, May 2003.

[2] G. Montenegro, N. Kushalnagar, J. Hui, D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, IETF, Sept. 2007. http://tools.ietf.org/html/rfc4944.

[3] Z. Shelby, K. Hartke, C. Bormann, B. Frank. Constrained Application Protocol (CoAP). draft-ietf-core-coap-18 (Work in Progress), IETF, May 2013. http://tools.ietf.org/html/draft-ietf-core-coap-18.

[4] E. Rescorla, N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, IETF, January 2012. http://tools.ietf.org/html/rfc6347.

[5] P. Eronen, H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279, IETF, December 2005. http://tools.ietf.org/html/rfc4279.

[6] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, T. Kivinen. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). draft-ietf-tls-oob-pubkey-10 (Work in Progress), IETF, October 19, 2013. http://tools.ietf.org/html/draft-ietf-tls-oob-pubkey-10.

[7] TelosB mote CM5000. [Online. Last accessed: 01.02.2014]. http://www.advanticsys.com/shop/mtmcm5000msp-p-14.html.

[8] L. Seitz, G. Selander. Additional Security Modes for CoAP. draft-seitz-core-security-modes-00 (Work in Progress), IETF, October 21, 2013. http://tools.ietf.org/html/draft-seitz-core-security-modes-00.

[9] SPDY Protocol - Draft 3. [Online. Last accessed: 04.02.2014]. http://www.chromium.org/spdy/spdy-protocol/sdit.

[11] A. Langley, W. Chang. Quick crypto. [Online. Last accessed: 04.02.2014]. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit.

[12] R. Zhang, Y. Zhang, and K. Ren, “Distributed Privacy-Preserving Access Control in Sensor Networks,” IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 8, pp. 1427–1438, 2012

[13] G. Zhang, W. Gong. The research of access control Based on UCON in the Internet of Things. Journal of Software, 6(4), April 2011.

[14] J. L. Hernandez-Ramos, A. J. Jara, L. Marin, A. F. Skarmeta. Distributed Capability-based Access Control for the Internet of Things. Journal of Internet Services and Information Security (JISIS), volume: 3, number: ¾, pp 1-16.

[15] Lightweight Machine to Machine. Technical specification (Work in Progress), Open Mobile Alliance, Jan 15, 2014.

[16] S. Gerdes, O. Bergmann, C. Bormann. Delegated CoAP Authentication and Authorization Framework (DCAF). draft-gerdes-core-dcaf-authorize-01 (Work in Progress), IETF, October 21, 2013. http://tools.ietf.org/html/draft-gerdes-core-dcaf-authorize-01.

[17] G. Selander, M. Sethi, L. Seitz. Access Control Framework for Constrained Environments. draft-selander-core-access-control-01 (Work in Progress), IETF, October 21, 2013. http://tools.ietf.org/html/draft-selander-core-access-control-01.

[18] M. Jones, J. Bradley, N. Sakimura. JSON Web Signature (JWS). draft-ietf-jose-json-web-signature-17 (Work in Progress), IETF, October 7, 2013. http://tools.ietf.org/html/draft-ietf-jose-json-web-signature-17.

[19] M. Jones, E. Rescorla, J. Hildebrand. JSON Web Encryption (JWE). draft-ietf-jose-json-web-encryption-17 (Work in Progress), IETF, October 7, 2013. http://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-17.

[20] M. Brown, R. Housley. Transport Layer Security (TLS) Authorization Extensions. RFC 5878, IETF, May 2010. http://tools.ietf.org/html/rfc5878.

[21] L. Seitz, G. Selander. Additional Security Modes for CoAP. draft-seitz-core-security-modes-00 (Work in Progress), IETF, October 21, 2013. http://tools.ietf.org/html/draft-seitz-core-security-modes-00.

[22] H. Krawczyk, M. Bellare, R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, IETF, February 1997. http://tools.ietf.org/html/rfc2104.

[23] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, IETF, October 2006. http://tools.ietf.org/html/rfc4648.

[24] STMicroelectronics. High-performance, IEEE 802.15.4 wireless system-on-chip with up
to 256 Kbytes of embedded Flash memory. September 2013. http://www.st.com/web/en/resource/technical/document/datasheet/CD00248316.pdf.

[25] Texas Instruments. CC2538 System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee®/ZigBee IP® Applications. April 2012–Revised May 2013. http://www.ti.com/lit/ug/swru319c/swru319c.pdf.

[26] Texas Instruments. CC2538EM Schematic Rev 1.2.0. December 02, 2012. http://www.ti.com/lit/zip/swrr113.

[27] Marius Ubostad. SmartRF06 Evaluation Board User's Guide. May 2013. http://www.ti.com/lit/ug/swru321a/swru321a.pdf.

[24] TinyOS. [Online. Last accessed: February 17 2014]. http://www.tinyos.net/.

[25] Contiki: The Open Source OS for the Internet of Things. [Online. Last accessed: February 17, 2014]. http://www.contiki-os.org/.

[26] Adam Dunkels, Bjorn. Gronvall, Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04), Tampa, FL, USA (Nov. 2004), pp. 455–462. http://dx.doi.org/10.1109/LCN.2004.38.

[27] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, Muneeb Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. SenSys'06, November 13, 2006, Boulder, Colorado, USA.

[28] Adam Dunkels. The ContikiMAC Radio Duty Cycling Protocol. SICS Technical Report T2011:13, ISSN 1100-3154, December 2004.

[29] Matthias Kovatsch, Simon Duquennoy, Adam Dunkels. A Low-Power CoAP for Contiki. IEEE IoTech 2011.

[30] The GnuTLS Transport Layer Security Library. [Online. Last accessed: February 17 2014]. http://gnutls.org/.

[31] MatrixSSL. [Online. Last accessed: February 17 2014]. http://matrixssl.org/.

[32] Olaf Bergmann. tinyDTLS. [Online. Last accessed: February 17 2014]. http://tinydtls.sourceforge.net/.

[33] List of battery sizes. [Online. Last accessed: February 17 2014]. http://en.wikipedia.org/wiki/List_of_battery_sizes

[34] ZigBee Document 053474r06, Version 1.0, ZigBee Specification. ZigBee Alliance. 2004.

[35] WirelessHART Technology. [Online. Last accessed: February 17 2014]. http://www.hartcomm.org/protocol/wihart/wireless_technology.html